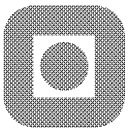
Norwegian University of Science and Technology
NTNU

Department of Computer and Information Science



FINAL EXAM [Eksamen]
TDT 4205 Compiler Technology [Kompilatorteknikk]
Thursday, December 13, 2007
Time: 09:00-13:00

Instructional contacts [faglige kontakter] during the exam:

Anne C. Elster (918 97 062) Jan Christian Meyer (411 93 264)

Aids [hjelpemidler]: C

No aids except information on assembler syntax, this year integrated in with this exam for you.

[Ingen hjelpemidler bortsett fra assembler-syntaks, i år skrevet inn på disse eksamensarkene for dere]

[Ikkje nokon hjelpemidlar tilet, da i år har eg skreve assemblersyntaksen inn i disse eksamensarka for dere]

Grades will be assigned within three working weeks.

[Karakterer vil bli satt innen tre arbeidsuker]
[Karekterar vert sette før tre airbeidsveker er omme]

ALL ANSWERS NEED TO BE WRITTEN ON THIS EXAM WHERE INDICATED. USE EXTRA SHEETS, IF NECESSARY, FOR CODING PROBLEMS.

[Alle svarene skal føres inn på oppgavearket der det er angitt plass, eventuelt på ekstra ark under kode oppgavene, hvis nødvendig.]

[Alle svara skal førast inn på oppgåvearket der det er gjeve plass, eventuelt på ekstra ark under kode oppgåva, om det trengs.]

It is NOT necessary to justify your answer on true/false questions.

[Det er ikke nødvendig å begrunne TRUE/FALSE spørsmål.]
[Ein treng ikkje grunngje TRUE/FALSE spørsmål.]

STUDENT NUMBER: _____

1. Warm-ups [oppvarming] – TRUE/ FALSE [Sant/u sant] (10/50)

Circle your answers -- Note: You will get a negative score for wrong answers and 0 for not answering or circling both TRUE and FALSE.

[Sett sirkel rundt svara -- NB: Dere får negative poeng om dere svarer feil, 0 poeng for å ikke/ikkje svare eller sirkle både "TRUE"(sant) og "FALSE" (usant).]

a) Lexers produce tokens ["lexere" lager "tokens"] TRUE/FALSE

b) Yacc produces tokens ["lexere" lager "tokens"] TRUE/FALSE

c) Syntax analysis is also known as parsing TRUE/FALSE
[syntaksanalyse er det samme som parsing]

d) Semantic analyzers use the symbol table and do type checking TRUE/FALSE
[semantiske analysatorer bruker symboltabeller og utfører typesjekk]

e) Syntax trees contain records for each variable name, with fields for the attributes of the name TRUE/FALSE
[syntakstrær inneholder "records" for hvert variabelnavn og felt for atributtene til navnet]

f) Precedence and associativity declarations make grammars ambigous. TRUE/FALSE
["Precedence"og "associativity declarations" gjør gramatikker tvetydige]

g) Top-down parsers do not need to figure out as much of the parse tree for a given amount of input compared to bottom-up parsers TRUE/FALSE

h) Heap variables don´t explicitly occur in low-level code TRUE/FALSE

i) C allows type aliases (C godtar "type aliases") TRUE/FALSE

j) Java allows type aliases TRUE/FALSE

1. CONTINUED (Fortsettelse av Oppg. 1):

k) new(Java) or malloc (C/C++) allocate variable space directly on stack TRUE/FALSE
[new(Java) og malloc(C/C++) alllokerer variabelplass direkte på stakken]

l) A frame (activation record) includes local variables TRUE/FALSE
[En "activation record" inkluderer locale variabler]

m) Dataflow analysis is not part of code optimization TRUE/FALSE
[dataflytanalyse er ikke en del av kodeoptimisering]

n) Derived induction variables are only incremented in loop body TRUE/FALSE
[utledete induksjonvariable er kun inkrementert innen løkkene]

o) Tiling w.r.t. instruction selection involves matching tree pattern to corresponding machine instruction TRUE/FALSE
["Tiling" m.h.t. valg av instruksjoner handler om å matche tremøstre til tilsvarende maskinkode]

p) It is reasonable to assume a roughly uniform tile cost when selecting instructions on a CISC machine TRUE/FALSE
[Er det rimelig å anta at en tilnærmet uniform tiling kostnad ved instruksjonsselektering på en CISC maskin?]

q) Dominators in CFGs are use to identify loops TRUE/FALSE
["Dominators" I kontrollflytgrafer er nyttet til å identifisere løkker]

r) Tree representation may be appropriate for instruction selection TRUE/FALSE
[Tree representasjoner kan være rett for instruksjonsutvalg.]

s) The Maximal Munch algorithm finds the least expensive tiling of a DAG TRUE/FALSE
["Maksimal Munch" algoritmer finner den minst dyre "tiling" av DAG]

t) The Pentium is a tree-address CISC architecture TRUE/FALSE
[Pentium er en tre-adresse CISC arkitektur]

2. **COMPILER BASICS. (10/50) In c)-j), fill in the blanks** (fyll inn svarene)

a) **Which of the following makes a compiler MORE complicated given Lex and Yacc?** [Hvilke av de følgende gjør en kompilator MER komplisert gitt Lex & Yacc?]
i) type checking
ii) objects
iii) ASTs
iv) NFA
v) REs (Regular Expressions)
vi) ambiguous grammars [tvetydige grammatikker]

b) **Which of the following can be recognized by a regular expression? Circle the right answer(s)**
[Hvilke av de følgende kan gjenkjennes av et regulært uttrykk? Sirkle svar(ene)]
i) input containing a string constant
ii) input containing nested function declarations
iii) input containing a floating point numeric constant
iv) order of identifier declarations [rekkefølge av identifikatordeklarasjoner]

c) **What is the difference between Lex and Yacc?** [Skilnaden på Lex og Yacc?]

---

d) **List two run-time checks done by a typical compiler:**
[To kjøretidssjekker gjort av en typisk kompilator?]
i) **ii)**

---

e) **How does Yacc handle a grammar that is ambiguous?**
[Hvordan håndterer Yacc tvetydige gramatikker?]

_____

f) **When do we need two pointers for the run-time stack (both stack and frame pointer)?**[Når trengs to pekere for kjøretidsstakken (både stakk og rammepeker?)]

_____

g) **What is strength reduction?** [Hva er "strength reduction"?]

_____

h) **How does a compiler type-check a method´s calling sequence (unlike a function)?** [Hvordan typesjekkes en metodes kallsekvens (i motsetning til funsjon)?]

_____

i) **Why do compilers NOT use the MOP (Meet Over Paths) solution rather than the MFP (Maximal Fixed Point) solution given that MOP is more precise?**

_____

j) **What is the problem with straightforward translation of low-IR code to assembly instructions on the target machine?**[Hva er roblemet med en direkte oversettelser av "low-IR"kode til assembler på målmaskinen?]

_____

3. Practical problems (15/50 points)

a) Related to PS1: Write an unambigious grammar for Palindromes over the alphabet {a, b}. (A palindrome is a word which spells the same both forwards and backwards, like "Otto").
[Skriv en utvetydig gramatikk for palindromer over alfabetet {a, b}]

b) Consider the following fragment from the Java grammar:
[Se på følgende fragment fra Javas gramatikk]

primary → f ield | call
field → **id** | primary . **id**
call → field parm
parm → ( ) | (primary)

Explain why this grammar is not suitable for a top-down parser.
[Forklar hvorfor denne gramatikken ikke er egnet for en Top-down parser.]

---

c) From PS 2: Calculate FIRST and FOLLOW for the nonterminals in the grammar below, and determine which nonterminals can derive ε [Regn ut FIRST og FOLLOW for "nonterminals" i følgende grammatikk samt si hvilke kan derivere ε]

S → uBDz
B → Bvlw
D → EF
E → yl ε
F → xl ε

	nullable?	FIRST	FOLLOW
S			
B			
D			
E			
F			

d) In PS3 you could have done some of the simplifications of the syntax trees directly in the parser (for slightly greater efficiency). Why did we not do that?

e) From PS4: Does a VSL program require any heap memory at runtime? Explain.

[fra PS4: Trenger et VSL program mine fra "heap" under kjøring? Forklar
]

---

3 f) Describe one advantage and one disadvantage with inheritance:
[Beskriv en fordel og en ulempe med arving ("inheritence")]

Advantage [fordel]: _____

Disadvantage [ulempe]:_____

g) Which variable(s) in the following C program may be eliminated by an optimizing compiler?

```
int f(int a,int b)
{
    int c[3],d,e;
    d=a+1;
    e=g(c,&b);
    return  e+c[1]+b;
}
```

Circle the variable(s) that may be eliminated: [Slå sirkel rundt eliminerbare variabler:]
a b c d e

h) Describe an example where loop unrolling would decrease performance:
[Beskriv et eksempel hvor utrulling av løkken vil svekke ytelsen]

i) Given the following program fragment [Gitt følgende programfragment:]

```
for (i=0 ; i<=100; i++)
    {
    a[2*i]=b[2*i];
    a[2*i+1]=-b[2*i+1];
    }
```

name the optimizations which produce : [Navngi optimaliseringene som produserer:]

```
for (i=0 ; i<=100; i++)
   {
    t=i+i;
    a[t]=b[t];
    a[t+1]=-b[t+1];
   }
```

Optimizations:_____

j) Is the precise cost of an instruction sequence easy to predict on a modern processor?
[Er den eksakte instruksjonskostnaden lett å beregne på moderne prosessorer?]

Why?/Why not? [Hvorfor?/Hvorfor ikke?]

_____

4. PROGRAMMING

Small instruction set reminder for IA-32: [Liten påminnelse om noen IA-32 instruksjoner:]

Instruction	Effect
mov %eax,(%esp)	- Move %eax into (%esp) [Flytt %eax til (%esp)]
push <src>	- Push value in src onto run-time stack [Skyv <src> på stakk]
pop <dst>	- Pop value from run-time stack into dst
add <src>,<dst>	- Add <src> to <dst> [Legg <src> til <dst>]
sub <src>,<dst>	- Subtract src from dst
neg <dst>	- Negate dst arithmetically
imul <src>	- Multiply 64-bit values in registers EDA, EAX by source (EDX contains higher order digits) [Gang 64-bit verdier I register EDA, EDX med <src> (EDX mest sigfikant)]
idiv <src>	- Divide 64-bit values in registers EDA, EAX by source (EDX contains higher order digits) [Del 64-bit verdiar I register EDA, EDX med <src> (EDX mest sigfikant)]

4a) Warm-up assember programming [3/50]

Re-write the following code fragment in IA-32 assembler (using "*GNU as*" syntax, just like we did on the Problem Sets) .The result should be left in the EAX register. Hint: read the next question

[omskriv følgende programsnutt til IA-32 assembler (med *"GNU as"* syntaks liksom programeringsoppgavene gitt tidligere i kurset). La resultatet være igjen in EAX registeret] Hint: Les neste oppgave

a = (a+b)/2

YOUR CODE[DIN CODE]:

4b) More programming [12/50] [Mer programmering]

Consider the scanner/parser program code below. This code forms a skeleton for parsing a simple language consisting of integer arithmetic expressions. It features a set of single-letter run-time variables, which are set as parameters to the resulting program. This allows the specification of programs which compute simple functions like the one in 4a) e.g. compiling (and assembling) the `text (a+b)/2` into an executable program "myfunc" that should result in an executable that gives the following output:

```
% ./myfunc 30 10
20
% ./myfunc 30 70
50
```

[se på scanner/parser programmet nedenfor. Denne koden gir et skjelett for parsing av et enkelt språk bestående av heltalls aritmetiske utrykk. Den tar i mot enkle bokstaver som kjørtidsvariabler. Disse er satt i hovedprogrammet. Dette tillater at spesfiksjoner av program som regner ut enkle funksjoner slik som den i 4a), f.eks. kompilasjon (og "assembling") av teksten (a+b)/2 til et eksekverbart program "myfunc" som gir resultatene ovenfor.]

Note that:

- **All arithmetic is 32-bit integer** [alle utregningene er 32-bit heltall]

- **Variables are single lowercase letters – 'a' refers to argument 1, 'b' to argument 2, etc**
  [variablene er enkle små bokstaver 'a' refererer til 1. argument, 'b' til 2. argument osv]

- **The provided macro ASM_HEAD gives assembly code to convert the program arguments into integers, and places them at offsets -4, -8, -12, … from the EBP register.**

[Gitt macro ASM_HEAD gir assemblerkode som oversetter programargumenter til heltall og putter dem ved "offsettene" -4, -8, -12 fra EBP register.]

- **The result of the evaluated expression must be left in the EAX register**
  [Resultatet av det evaluerte uttrykket skal bli liggende I EAX registeret]

Your task is to fill in the missing part of the code in order to complete such an expression compiler, which targets IA-32 assembly language (using "GNU as" syntax).

You should complete the ASM_TAIL macro such that the program outputs the result value using `printf` before terminating.

[Din oppgave er å fylle in de manglende kodesnuttene for en slik uttrykkskompilator som er rettet mot IA-32 assembler (med "GNU as" syntaks).
Du bør fullføre ASM_TAIL makroen slik at programmet gir ut reultatverdiene ved bruk av `printf` før det terminerer.]

```
/*
 * calculator_primitive.l
 * Scanner for a simple expression compiler in Lex / Yacc
 */

%{
    #include "calculator_primitive.tab.h"
%}
%option noyywrap
%%
[\ \t]+  {}
\n          { return NEWLINE; }
[a-z]       { yylval = yytext[0]; return IDENTIFIER; }
[0-9]+      { yylval = strtol( yytext, NULL, 10 ); return INTEGER; }
.           { return yytext[0]; }
%%


/*
 * calculator_primitive.y
 * Parser for a simple expression compiler in Lex / Yacc
 */

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ASM_HEAD \
".data\n"\
"FORMAT_STRING:\n"\
".string \"%d\\n\"\n"\
".globl main\n"\
"main:\n"\
"   pushl %%ebp\n"\
"   movl %%esp,%%ebp\n"\
"   movl 8(%%esp),%%esi /* Store the first parameter (argc) in ESI */\n"\
"   decl %%esi          /* argc--; argv[0] is not interesting to us */\n"\
"   jz   noargs         /* Skip argument setup if there are none */\n"\
"\n"\
"   movl    12(%%ebp),%%ebx /* Store the base addr. of argv in EBX */\n"\
"pusharg:                   /* Loop over the arguments */\n"\
"   addl $4,%%ebx     /* Look at the next argument (disregarding argv[0])
*/\n"\
"   pushl $10        /* strtol arg 3: our number base is 10 */\n"\
"   pushl $0         /* strtol arg 2: there is no error pointer */\n"\
"   pushl (%%ebx)    /* strtol arg 1: Addr. of string containing integer
*/\n"\
"   call strtol    /* Call strtol, to convert the string to a 32-bit int
*/\n"\
"   addl $12,%%esp /* Restore the stack pointer to before strtol-params
*/\n"\
"   pushl %%eax    /* Push return value from strtol (our new argument)
*/\n"\
"   decl %%esi     /* Decrement the number of arguments to go */\n"\
"   jnz pusharg    /* Loop if there are any arguments left */\n"\
"noargs:\n"
```

```
#define ASM_TAIL \
"                               "\

"                               "\

"                               "\

"                               "\

"                               "\

"   leave\n"\
"   ret\n"
%}

%token INTEGER IDENTIFIER NEWLINE

/* Define operator precedence and associativity. */
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
function: expr NEWLINE { printf ( "    popl %%eax\n" );};
expr:
      expr '+' expr
      {









      }
    | expr '-' expr
      {









      }
    | expr '*' expr
      {









      }
    | expr '/' expr
      {









      }
```

```
    | '(' expr ')'
      {


      }
    | '-' expr %prec UMINUS
      {


      }
    | IDENTIFIER
      {


      }
    | INTEGER
      {


      }
    ;
%%

/* This definition is required by bison */
int
yyerror ( void )
{
    fprintf ( stderr, "Syntax error\n" );
    return 1;
}


/* Our main function - translate an expression and quit */
int
main ( int argc, char **argv )
{
    printf ( ASM_HEAD );
    yyparse();
    printf ( ASM_TAIL );
    exit ( EXIT_SUCCESS );
}
```